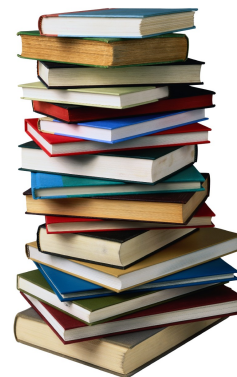


Pilhas

Uma pilha é uma estrutura de dados que admite remoção de elementos e inserção de novos elementos. Mais especificamente, uma pilha (= *stack*) é uma estrutura sujeita à seguinte regra de operação: sempre que houver uma remoção,

o elemento removido é o que está na estrutura há menos tempo.

Em outras palavras, o primeiro objeto a ser inserido na pilha é o último a ser removido. Essa política é conhecida pela sigla LIFO (= *Last-In-First-Out*).



Implementação em um vetor

Suponha que nossa pilha está armazenada em um vetor `pilha[0..N-1]`. (A natureza dos elementos do vetor é irrelevante: eles podem ser inteiros, caracteres, ponteiros, etc.) Digamos que a parte do vetor ocupada pela pilha é

```

pilha[0..t-1] .

```

O índice t indica o *topo* da pilha, de modo que $\text{pilha}[t]$ é a primeira posição vaga da pilha. A pilha está *vazia* se t vale 0 e *cheia* se t vale N . No exemplo da figura, os números 111, 222, ..., 888 foram inseridos na pilha nessa ordem:

0							t				N-1
111	222	333	444	555	666	777	888				

Para **remover**, ou **tirar**, um elemento da pilha — esta operação é conhecida como *desempilhar* (= *to pop*) — faça

```
x = pilha[--t];
```

Isso equivale ao par de instruções `"t -= 1; x = pilha[t];"`, nesta ordem. É claro que você só deve desempilhar se tiver certeza de que a pilha não está vazia. Para **consultar** a pilha sem desempilhar, faça `x = pilha[t-1]`.

Para *inserir*, ou *colocar*, um objeto y na pilha — a operação é conhecida como *empilhar* (= *to push*) — faça

```

pilha[t++] = y;

```

Isso equivale ao par de instruções `"pilha[t] = y; t += 1;"`, nesta ordem. Antes de empilhar, verifique se a pilha já está cheia para evitar que ela *transborde*, ou seja, para evitar um *overflow*. Em geral, a tentativa de inserir em uma pilha cheia é uma situação excepcional, que indica um mau planejamento lógico do seu programa.

Exercícios 1

1. Suponha que, diferentemente da convenção adotada no texto, a parte do vetor ocupada pela pilha é `pilha[0..t]`. Escreva a instrução que retira um elemento da pilha. Escreva a instrução que coloca um objeto `y` na pilha.
2. Escreva funções `desempilha` e `empilha` para manipular uma pilha. Lembre-se de que uma pilha é um pacote com dois objetos: um vetor e um índice. Quais os parâmetros de suas funções? Não use variáveis globais.

Aplicação: parênteses e colchetes

Considere o problema de decidir se uma dada [sequência](#) de parênteses e colchetes está bem-formada (ou seja, parênteses e colchetes são fechados na ordem inversa àquela em que foram abertos). Por exemplo, a primeira das sequências abaixo está bem-formada enquanto a segunda não está:

(() [()]) ([]]

Suponha que a sequência de parênteses e colchetes está armazenada em uma [string](#) (uma cadeia de caracteres) `s`. Como é hábito em C, o último caractere da string é `'\0'` (o [caractere nulo](#)).

```
// Esta função devolve 1 se a string s contém uma
// sequência bem-formada de parênteses e colchetes
// e devolve 0 se a sequência é malformada.

int bemFormada( char s[]) {
    char *pilha; int t;
    int N, i;

    N = strlen( s);
    pilha = malloc( N * sizeof (char));
    t = 0;
    for (i = 0; s[i] != '\0'; ++i) {
        // a pilha está armazenada no vetor pilha[0..t-1]
        switch (s[i]) {
            case '(': if (t != 0 && pilha[t-1] == '(')
                        --t;
                      else return 0;
                      break;
            case '[': if (t != 0 && pilha[t-1] == '[')
                        --t;
                      else return 0;
                      break;
            default: pilha[t++] = s[i];
        }
    }
    return t == 0;
}
```

(Eu deveria ter invocado `free(pilha)` antes de cada `return`. Só não fiz isso para não obscurecer a lógica da função.) Note que a pilha não transborda porque nunca terá mais elementos que o número de caracteres de `s`.

Exercícios 2

1. A função `bemFormada` funciona corretamente se `s` tem apenas dois elementos? apenas um? nenhum?
2. Considere a função `bemFormada`. Mostre que no início de cada iteração `s` está bem-formada se e somente se a sequência `pilha[0..t-1] s[i...]` estiver bem-formada.
3. Escreva uma versão melhorada da função `bemFormada` que desaloque o vetor `pilha` antes de terminar a execução da função. [\[Solução\]](#)

Outra aplicação: notação polonesa

Na notação usual de expressões aritméticas, os operadores são escritos *entre* os operandos; por isso, a notação é chamada *infixa*. Na notação *polonesa*, ou *posfixa*, os operadores são escritos *depois* dos operandos. Eis alguns exemplos de expressões infixas e correspondentes expressões posfixas:

infixa	posfixa
$(A+B*C)$	$ABC*+$
$(A*(B+C)/D-E)$	$ABC+*D/E-$
$(A+B*(C-D*(E-F)-G*H)-I*3)$	$ABCDEF-* -GH* -*+I3*-$
$(A+B*C/D*E-F)$	$ABC*D/E*+F-$
$(A+(B-(C+(D-(E+F)))))$	$ABCDEF+-+--+$
$(A*(B+(C*(D+(E*(F+G))))))$	$ABCDEFG+*****$

Note que a ordem em que os operandos (A, B, C, etc.) aparecem numa expressão posfixa é a mesma da expressão infix correspondente. Note também que a notação posfixa dispensa parênteses e dispensa as [regras de precedência entre operadores](#) (como a precedência de * sobre + por exemplo) que são indispensáveis na notação infix.

Nosso problema: traduzir para notação posfixa a expressão infix armazenada em uma string `inf`. Para simplificar nossa vida, vamos supor que a expressão infix está correta e consiste apenas em letras, abreviações, parênteses, e símbolos para as quatro operações aritméticas. Vamos supor também que os nomes das variáveis têm apenas uma letra cada. Finalmente, vamos supor que a expressão toda está embrulhada em um par de parênteses, isto é, o primeiro elemento de `inf` é '(' e os dois últimos são ') ' e '\0'.

Usaremos uma pilha para resolver o problema. Como a expressão infix está embrulhada em um par de parênteses, não precisamos nos preocupar com pilha vazia!

```
// Esta função abaixo recebe uma expressão infix inf
// e devolve a correspondente expressão posfixa.

char *infixaParaPosfixa( char inf[]) {
    char *posf;
    char *pi; int t; // pilha
    int N, i, j;

    N = strlen( inf);
    posf = malloc( N * sizeof (char));
    pi = malloc( N * sizeof (char));
    t = 0;
    pi[t++] = inf[0]; // empilha '('
    for (j = 0, i = 1; /*PP*/ inf[i] != '\0'; ++i) {
        // a pilha está em pi[0..t-1]
        switch (inf[i]) {
            char x;
            case '(': pi[t++] = inf[i]; // empilha
                    break;
            case ')': while (1) { // desempilha
                        x = pi[--t];
                        if (x == '(') break;
                        posf[j++] = x;
                    }
                    break;
            case '+':
            case '-': while (1) {
                        x = pi[t-1];
                        if (x == '(') break;
                        --t; // desempilha
                        posf[j++] = x;
                    }
                    pi[t++] = inf[i]; // empilha
                    break;
            case '*':
            case '/': while (1) {
                        x = pi[t-1];
                        if (x == '(' || x == '+' || x == '-')
                            break;
                        --t;
                        posf[j++] = x;
                    }
                    pi[t++] = inf[i];
                    break;
            default: posf[j++] = inf[i];
        }
    }
    free( pi);
    posf[j] = '\0';
    return posf;
}
```

(O código pode ser [escrito de maneira um pouco diferente](#) se tirarmos proveito dos recursos sintáticos da linguagem C.) Constantes e variáveis vão diretamente de `inf` para `posf`. Todo abre-parêntese é colocado na pilha. Ao encontrar um fecha-parêntese, a função remove tudo da pilha até um abre-parêntese inclusive. Ao encontrar um `+` ou um `-`, a função desempilha tudo até um abre-parêntese exclusive. Ao encontrar um `*` ou um `/`, a função desempilha tudo até um abre-parêntese ou um `+` ou um `-`.

Eis o resultado da aplicação da função à expressão infixa $(A * (B * C + D))$. A tabela abaixo registra os valores das variáveis a cada passagem pelo ponto `pp` do código:

<u>inf[0..i-1]</u>	<u>pi[0..t-1]</u>	<u>posf[0..j-1]</u>
((
(A	(A
(A*	(*	A
(A* ((* (A
(A* (B	(* (AB
(A* (B*	(* (*	AB
(A* (B*C	(* (*	ABC
(A* (B*C+	(* (+	ABC*
(A* (B*C+D	(* (+	ABC*D
(A* (B*C+D)	(*	ABC*D+
(A* (B*C+D))		ABC*D+*

Exercícios 3

- Use o algoritmo para converter a expressão infixa $(A + B) * D + E / (F + A * D) + C$ na expressão posfixa equivalente.
- Considere a função [infixaParaPosfixa](#). Suponha que a expressão infixa `inf` tem `N` caracteres (sem contar o `'\0'` final). Que tamanho a pilha `pi` pode atingir, no pior caso? Em outras palavras, qual o valor máximo da variável `t` no pior caso? Que acontece se o número de abre-parênteses na expressão for limitado (menor que 6, por exemplo)?
- No código da função [infixaParaPosfixa](#), alguns casos têm um `"while (1)"`. Escreva uma nova versão sem esses `"while"`. (Dica: troque o `for` externo por um `while` apropriado.)
- Reescreva a função [infixaParaPosfixa](#) sem supor que a expressão infixa está embrulhada em um par de parênteses.
- Reescreva a função [infixaParaPosfixa](#) supondo que a expressão pode ter *colchetes* além de parênteses.
- Reescreva a função [infixaParaPosfixa](#) supondo que a expressão pode não estar bem-formada.

Exercícios 4

- VALOR DE EXPRESSÃO POLONESA. Suponha que `posf` é uma string que guarda uma expressão aritmética em notação posfixa. Suponha que `posf` não é vazia e contém somente os operadores `+`, `-`, `*` e `/` (todos exigem dois operandos). Suponha também que a expressão não tem constantes e que todos os nomes de variáveis na expressão consistem em uma única letra maiúscula. Suponha ainda que temos um vetor `tabela` que dá os valores das variáveis (todas os valores são inteiros):

`tabela[0]` é o valor da variável `A`,
`tabela[1]` é o valor da variável `B`, etc.

Escreva uma função que calcule o valor da expressão `posf`. Cuidado com divisões por zero!

- Escreva um algoritmo que use uma pilha para *inverter a ordem* das letras de cada palavra de uma string, preservando a ordem das palavras. Por exemplo, dado o texto `ESTE EXERCICIO E MUITO FACIL` a saída deve ser `ETSE OICICREXE E OTIUM LICAF`.
- Digamos que nosso alfabeto contém apenas as letras `a`, `b` e `c`. Considere o seguinte conjunto de strings sobre

nosso alfabeto:

c, aca, bcb, abcba, bacab, aacaa, bbcbb, ...

Qualquer string desse conjunto tem a forma WcM , onde W é uma sequência de letras que só contém a e b e M é o inverso de W , ou seja, M é W lido de trás para frente. Escreva um programa que determina se uma string X pertence ou não ao nosso conjunto, ou seja, determina se X é da forma WcM .

4. PERMUTAÇÕES PRODUZIDAS PELO DESEMPILHAR. Suponha que os inteiros 1,2,3,4 são colocados, nesta ordem, numa pilha inicialmente vazia. Depois de empilhar cada inteiro, você pode retirar zero ou mais elementos da pilha. Cada elemento desempilhado é impresso numa folha de papel. Por exemplo, a sequência de operações empilha 1, empilha 2, desempilha, empilha 3, desempilha, desempilha, empilha 4, desempilha, produz a impressão da sequência 2,3,1,4. Quais das 24 permutações de 1,2,3,4 podem ser obtidas desta maneira?

Pilha implementada em uma lista encadeada

Como implementar uma pilha em uma [lista encadeada](#)? Digamos que as células da lista são do tipo `celula`:

```
typedef struct cel {
    int      conteudo;
    struct cel *prox;
} celula;
```

Decisões de projeto: Vamos supor que nossa lista tem uma célula-cabeça (e portanto a primeira célula da lista não faz parte da pilha). Vamos supor também que o topo da pilha está na *segunda* célula e não na última (por quê?). A pilha pode ser criada e inicializada assim:

```
celula cabeca;
celula *tp;
tp = &cabeca; // topo da pilha
tp->prox = NULL;
```

Para ter acesso à pilha, só preciso do ponteiro `tp`. De acordo com nossa decisão de projeto, teremos sempre `tp == &cabeca`. A pilha está *vazia* se `tp->prox == NULL`.

```
// Coloca y na pilha tp.

void empilha( int y, celula *tp) {
    celula *nova;
    nova = malloc( sizeof (celula));
    nova->conteudo = y;
    nova->prox = tp->prox;
    tp->prox = nova;
}

// Remove um elemento da pilha tp
// e devolve o elemento removido.
// Supõe que a pilha não está vazia.

int desempilha( celula *tp) {
    int x;
    celula *p;
    p = tp->prox;
    x = p->conteudo;
    tp->prox = p->prox;
    free( p);
    return x;
}
```

Exercícios 5

1. Implemente um pilha em uma lista encadeada *sem* célula-cabeça (só pra ver ver que dor de cabeça isso dá!). A pilha será dada pelo endereço da primeira célula da lista (que é também o topo da pilha).
2. Reescreva as funções [bemFormada](#) e [infixaParaPosfixa](#) armazenando a pilha em uma lista encadeada.
3. Resolva o problema da [intercalação](#) de duas listas ordenadas.
4. Suponha dada uma lista encadeada que armazena números inteiros. Escreva uma função que transforme a lista em duas: a primeira contendo as células cujo conteúdo é par e a segunda aquelas cujo conteúdo é ímpar.

Apêndice: A pilha de execução de um programa

Para executar um programa, o computador usa uma **pilha de execução**. (Veja o verbete [Call stack](#) na Wikipedia.) A operação pode ser descrita conceitualmente da seguinte maneira:

Todo programa C consiste em uma ou mais funções (sendo `main` a primeira função a ser executada). Ao encontrar a invocação de uma função, o computador cria um novo "espaço de trabalho", que contém todos os parâmetros e todas as variáveis locais da função. Esse espaço de trabalho é colocado na pilha de execução (por cima do espaço de trabalho que invocou a função) e a execução da função começa (confinada ao seu espaço de trabalho). Quando a execução da função termina, o seu espaço de trabalho é removido da pilha e descartado. O espaço de trabalho que estiver agora no topo da pilha é reativado e a execução é retomada do ponto em que havia sido interrompida.

Considere o seguinte exemplo:

```
int G( int a, int b) {
    int x;
    x = a + b;
    return x;
}

int F( int i, j, k) {
    int x;
    x = /*2*/ G( i, j) /*3*/;
    x = x + k;
    return x;
}

int main( void) {
    int i, j, k, y;
    i = 111; j = 222; k = 444;
    y = /*1*/ F( i, j, k) /*4*/;
    printf( "%d\n", y);
    return EXIT_SUCCESS;
}
```

A execução do programa prossegue da seguinte maneira:

- Um espaço de trabalho é criado para a função `main` e colocado na pilha de execução. O espaço contém as variáveis locais `i`, `j`, `k` e `y`. A execução de `main` começa.
- No ponto 1, a execução de `main` é temporariamente interrompida e um espaço de trabalho para a função `F` é colocado na pilha. Esse espaço contém os parâmetros `i`, `j`, `k` da função (com valores 111, 222 e 444 respectivamente) e a variável local `x`. Começa então a execução de `F`.
- No ponto 2, a execução de `F` é interrompida e um espaço de trabalho para a função `G` é colocado na pilha. Esse espaço contém os parâmetros `a` e `b` da função (com valores 111 e 222 respectivamente) e a variável local `x`. Em seguida, começa a execução de `G`.
- Quando a execução de `G` termina, a função devolve 333. O espaço de trabalho de `G` é removido da pilha e descartado. O espaço de trabalho de `F` (que agora está no topo da pilha de execução) é reativado e a execução é retomada no ponto 3. A primeira instrução executada é "`x = 333;`".
- Quando a execução de `F` termina, a função devolve 777. O espaço de trabalho de `F` é removido da pilha e descartado. O espaço de trabalho de `main` (que agora está no topo da pilha) é reativado e a

execução é retomada no ponto 4. A primeira instrução executada é " $y = 777$ ";".

No nosso exemplo, F e G são funções distintas. Mas tudo funcionaria da mesma maneira se F e G fossem idênticas, ou seja, se F fosse uma função recursiva.

Exercícios 6

1. Considere a função recursiva abaixo. Escreva uma versão iterativa da função que simule o comportamento da versão recursiva. Use uma pilha.

```
int TTT( int x[], int n) {  
    if (n == 0) return 0;  
    if (x[n] > 0) return x[n] + TTT( x, n-1);  
    else return TTT( x, n-1);  
}
```

Veja o verbete [Stack \(data structure\)](#) na Wikipedia

Last modified: Mon Apr 13 07:12:53 BRT 2015

<http://www.ime.usp.br/~pf/algoritmos/>

Paulo Feofiloff

[DCC-IME-USP](#)

